Digital Design Term Project 2

Moon Gi Jun 2013210063

정주혜 2015320142

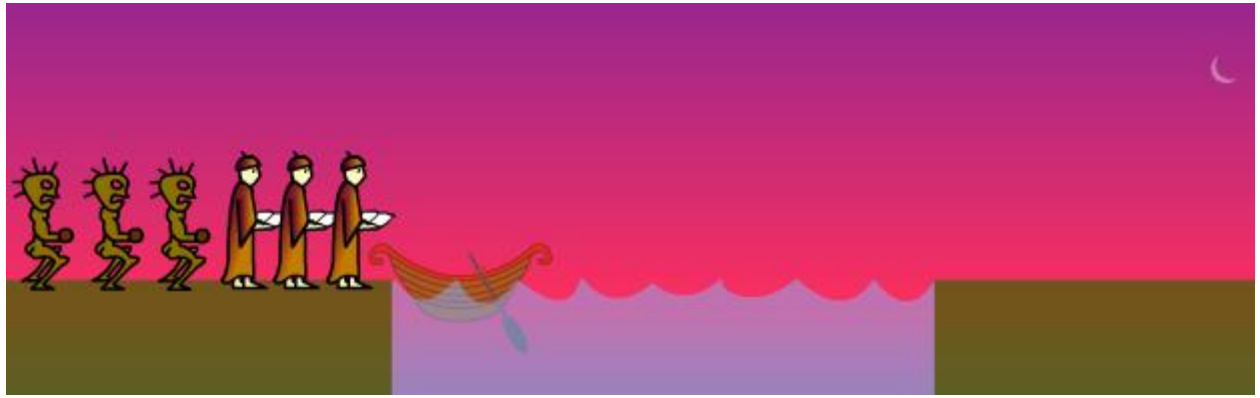
목차

Project 요약

1. 선교사와 식인종 문제란?
2. Project의 목적

선교사와 식인종 문제란?

- 고전적인 퀴즈 문제로서 컴퓨터 분야에서는 인공 지능의 한 문제
- 3명의 선교사와 3명의 식인종이 나룻배를 타고 강을 건너려고 하는데, 나룻배에는 2명 밖에는 탈 수 없다. 만일 강의 어느 쪽 에서라도 식인종의 수가 선교사의 수보다 많으면 식인종들은 선교사들을 잡아먹게 된다.
- 이 때 선교사들이 잡혀 먹히지 않고 6명이 무사히 강을 건너려면 어떻게 해야 하는가 하는 문제이다.



Project의 목적(1차과제의 조합회로)

- 오른쪽에 나와 있는 테이블은 강을 왼쪽의 선교사와 식인종의 수, 그리고 어떤 방향으로 배가 누구를 몇 명 태우고 움직였는지 , 그리고 강을 건넌 선교사와 식인종의 수를 차례대로 나타낸다.
- 이 테이블을 이용하여 다음 과 같이 강 왼쪽 상태에 대한 데이터를 만들 수 있다.

차례	현재 상태	이동 방향	다음 상태
1	11 11	1	11 01
2	11 01	0	11 10
3	11 10	1	11 00
..	..	..	..

- 이 테이블을 이용하여 gate level 로 조합 회로를 구현하는 것이 이 과제의 문제이다. 구현에 있어 몇가지 조건이 있는데 그것은:
  - 1. 현재 상태가 마지막 상태 (모두 건너 감)인 경우,
    다음 상태는 초기 상태 (모두 건너가지 않음)로 돌아간다.
  - 2. 현재 상태가 해답과 거리가 먼 경우, 마찬가지로
    다음 상태는 초기 상태 (모두 건너가지 않음)로 돌아간다.
- 이 데이터를 이용하여 5개의 Input과 3개의 Output으로 구성된 조합 회로를 K-map을 통하여 간략화시키고 그것을 HDL code를 이용하여 Quartus 를 통해 구현한다.

차례	왼쪽	배	오른쪽
0	MMM CCC		
1	MMMC	CC ->	CC
2	MMMCC	<- C	C
3	MMM	CC ->	CCC
4	MMMC	<- C	CC
5	MC	MM ->	MM CC
6	MMCC	<- MC	M C
7	CC	MM ->	MMM C
8	CCC	<- C	MMM
9	C	CC ->	MMM CC
10	CC	<- C	MMM C
11		CC ->	MMM CCC

Project의 목적(2차과제의 추가사항)

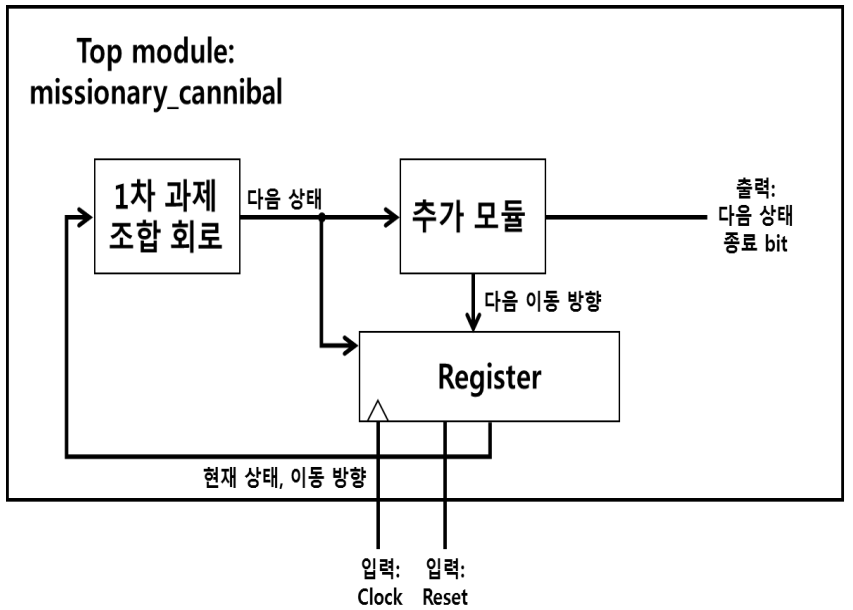
- 1차 과제와 다르게 Clock과 reset을 입력으로 받는다. Reset이 1일 때, 초기 상태 (11 11 1)가 register로 load되어야 한다. 또한 다음 상태와 종료를 알리는 1 bit를 출력으로 낸다. 여기서 register는 D플립플랍으로 구성되며 이전 상태를 받아 기억하는 순차회로이다.
- 종료를 알리는 1bit는 최종 상태에 도달했을 때에만 종료를 알리는 bit가 1로 출력된다.
- 다음 상태는 1차 과제에서 만든 조합 회로의 출력과 동일하며 최종 상태에 도달한 경우, 다음 상태는 초기 상태로 돌아간다.

* 이 그래프와 같이 말이다.

Reset으로 초기 상태를 register에 load함

Clock	현재 상태	이동 방향	다음 상태	종료
1	11 11	1	11 01	0
2	11 01	0	11 10	0
3	11 10	1	11 00	0
..	..	..	..	..
10	00 01	0	00 10	0
11	00 10	1	00 00	1
12	00 00	0	11 11	0
13	11 11	1	11 01	0
..	..	..	..	..

최종 상태 도달 시, 초기 상태로 돌아감



*회로에 대한 특성을 그리자면 위의 그림과 같다.

- 1차 과제와 같이 현재 상태가 해답과 거리가 먼 경우, 다음 상태는 초기 상태로 돌아간다. 그러나, 이와 같은 상황이 발생하지 않도록 구현해야 한다. 즉, 해답이 Clock과 Reset의 인풋에 의해 순서대로 변화해야 한다.
- 결과적으로 오른쪽 그림과 같은 회로를 HDL code를 이용하여 Quartus 를 통해 구현하는 것이 이 Project의 목적이다.

Module 설명

I. 1차 과제 조합회로 모듈의 구성

II. 추가 모듈의 구성

III. Register (여러 개의 D FlipFlop)의 구성

IV. 헤더 모듈의 구성

V. Clock Cycle Time을 줄이는 방법

1차과제 조합회로 모듈의 구성

A. Module 구성을 위한 Truth Table
B. Truth Table을 기반으로 한 K-map
C. K-map을 이용한 gate level 구조 파악
D. HDL Code

Module 구성을 위한 Truth Table

1. 전체 Input Output 을 포함한 Truth Table

	Input					Output				
	현재 상태				이동 방향	다음 상태				
순서	missionary_curr[0]	missionary_curr[1]	cannibal_curr[0]	cannibal_curr[1]	direction	missionary_next[0]	missionary_next[1]	cannibal_next[0]	cannibal_next[1]	차례
0	0	0	0	0	0	1	1	1	1	
1	0	0	0	0	1	1	1	1	1	12
2	0	0	0	1	0	0	0	1	0	10
3	0	0	0	1	1	1	1	1	1	
4	0	0	1	0	0	0	0	1	1	8
5	0	0	1	0	1	0	0	0	0	11
6	0	0	1	1	0	1	1	1	1	
7	0	0	1	1	1	0	0	0	1	9
8	0	1	0	0	0	1	1	1	1	
9	0	1	0	0	1	1	1	1	1	
10	0	1	0	1	0	1	0	1	0	6
11	0	1	0	1	1	1	1	1	1	
12	0	1	1	0	0	1	1	1	1	
13	0	1	1	0	1	1	1	1	1	
14	0	1	1	1	0	1	1	1	1	
15	0	1	1	1	1	1	1	1	1	
16	1	0	0	0	0	1	1	1	1	
17	1	0	0	0	1	1	1	1	1	
18	1	0	0	1	0	1	1	1	1	
19	1	0	0	1	1	1	1	1	1	
20	1	0	1	0	0	1	1	1	1	
21	1	0	1	0	1	0	0	1	0	7
22	1	0	1	1	0	1	1	1	1	
23	1	0	1	1	1	1	1	1	1	
24	1	1	0	0	0	1	1	0	1	4
25	1	1	0	0	1	1	1	1	1	
26	1	1	0	1	0	1	1	1	0	2
27	1	1	0	1	1	0	1	0	1	5
28	1	1	1	0	0	1	1	1	1	
29	1	1	1	0	1	1	1	0	0	3
30	1	1	1	1	0	1	1	1	1	
31	1	1	1	1	1	1	1	0	1	1

2. Direction 0의 Truth Table

Direction 이 "0"인 Truth Table									
순서	missionary_curr[0]	missionary_curr[1]	cannibal_curr[0]	cannibal_curr[1]	missionary_next[0]	missionary_next[1]	cannibal_next[0]	cannibal_next[1]	차례
0	0	0	0	0	1	1	1	1	12
1	0	0	0	1	0	0	1	0	10
2	0	0	1	0	0	0	1	1	8
3	0	0	1	1	1	1	1	1	
4	0	1	0	0	1	1	1	1	
5	0	1	0	1	1	0	1	0	6
6	0	1	1	0	1	1	1	1	
7	0	1	1	1	1	1	1	1	
8	1	0	0	0	1	1	1	1	
9	1	0	0	1	1	1	1	1	
10	1	0	1	0	1	1	1	1	
11	1	0	1	1	1	1	1	1	
12	1	1	0	0	1	1	0	1	4
13	1	1	0	1	1	1	1	0	2
14	1	1	1	0	1	1	1	1	
15	1	1	1	1	1	1	1	1	

3. Direction 1의 Truth Table

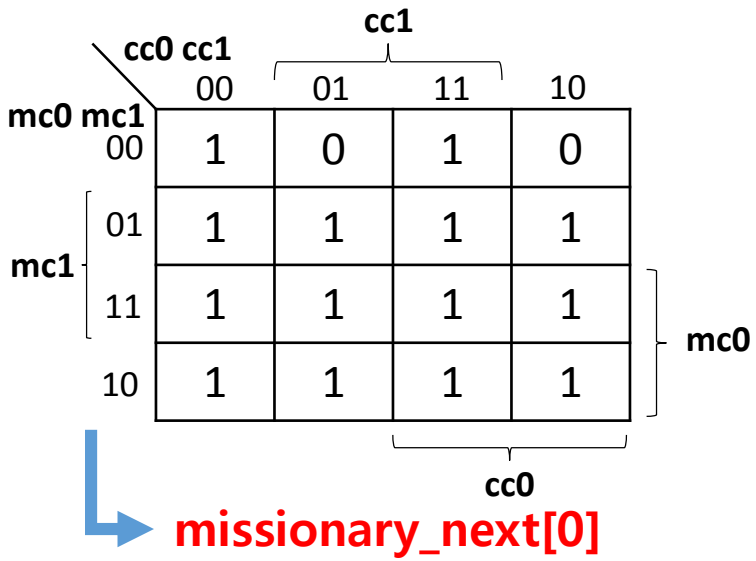
Direction 이 "1"인 Truth Table									
순서	missionary_curr[0]	missionary_curr[1]	cannibal_curr[0]	cannibal_curr[1]	missionary_next[0]	missionary_next[1]	cannibal_next[0]	cannibal_next[1]	차례
1	0	0	0	0	1	1	1	1	
3	0	0	0	1	1	1	1	1	
5	0	0	1	0	0	0	0	0	11
7	0	0	1	1	0	0	0	1	9
9	0	1	0	0	1	1	1	1	
11	0	1	0	1	1	1	1	1	
13	0	1	1	0	1	1	1	1	
15	0	1	1	1	1	1	1	1	
17	1	0	0	0	1	1	1	1	
19	1	0	0	1	1	1	1	1	
21	1	0	1	0	0	0	1	0	7
23	1	0	1	1	1	1	1	1	
25	1	1	0	0	1	1	1	1	
27	1	1	0	1	0	1	0	1	5
29	1	1	1	0	1	1	0	0	3
31	1	1	1	1	1	1	0	1	1

Truth Table을 기반으로 한 K-map

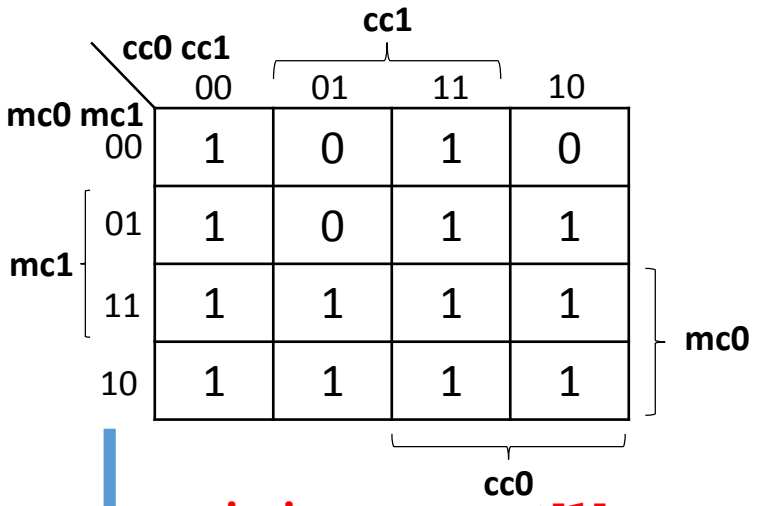
1. Direction 0 의 K-map
2. Directrion 1의 K-map
3. missionary_next[0] 의 3차원 K-map
4. missionary_next[1] 의 3차원 K-map
5. cannibal_next[0] 의 3차원 K-map
6. cannibal_next[1] 의 3차원 K-map

Direction 0의 K-map

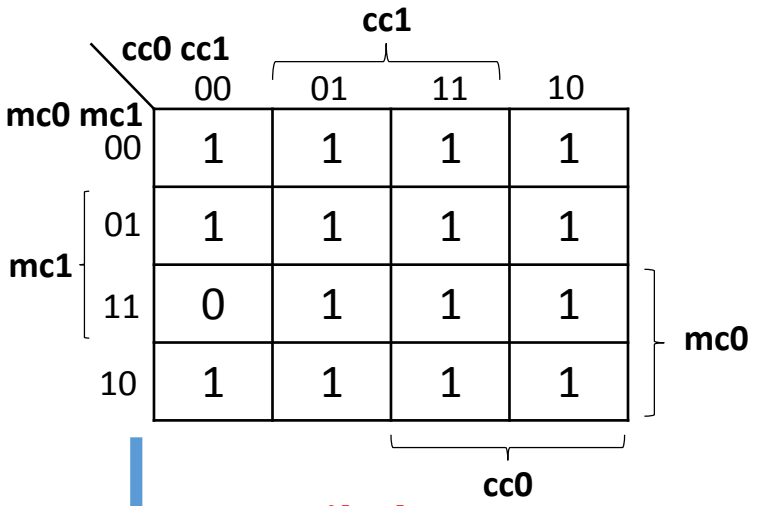
missionary_curr[0] = mc0
missionary_curr[1]= mc1
cannibal_curr[0] = cc0
cannibal_curr[1] = cc1



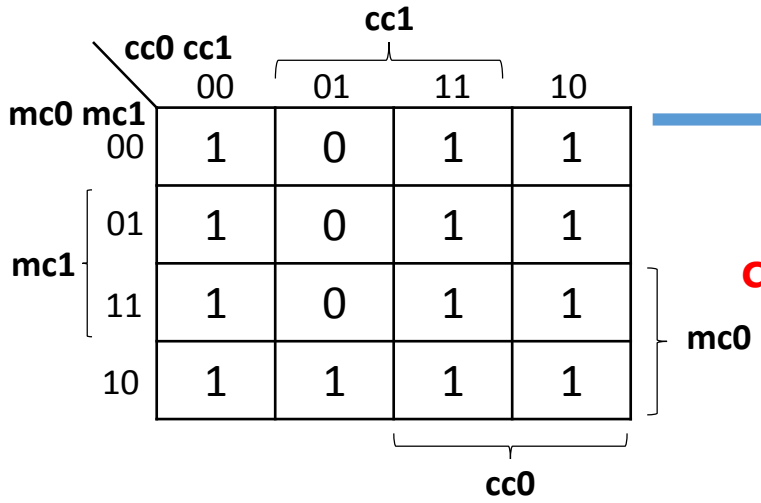
missionary_next[0]



missionary_next[1]



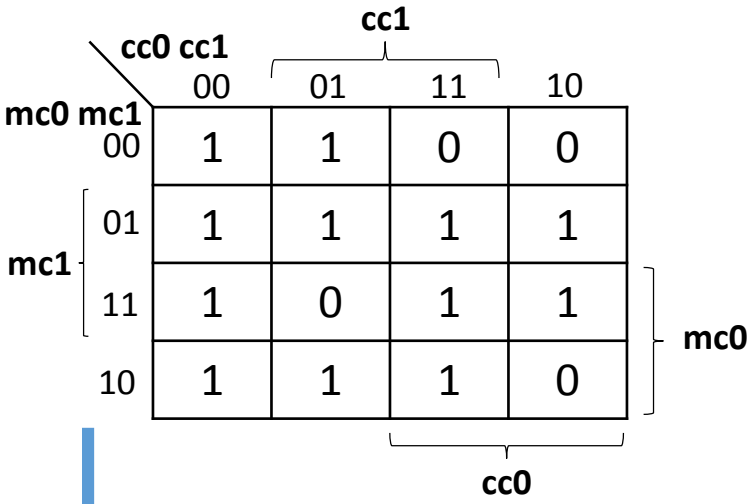
cannibal_next[0]



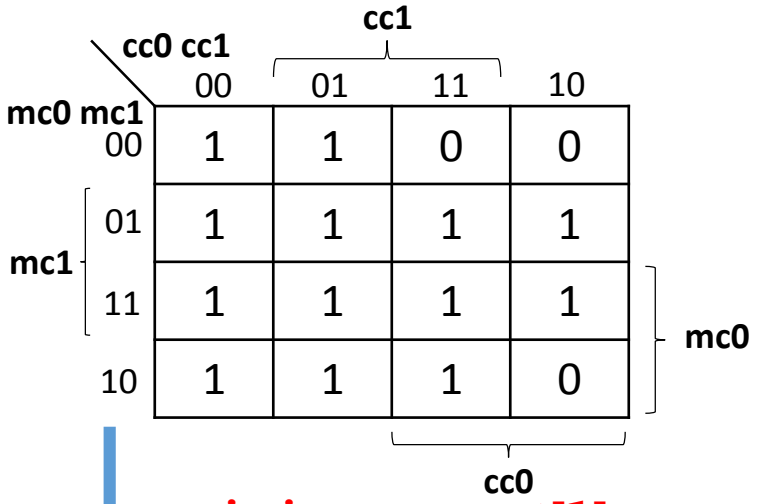
cannibal_next[1]

Direction 1의 K-map

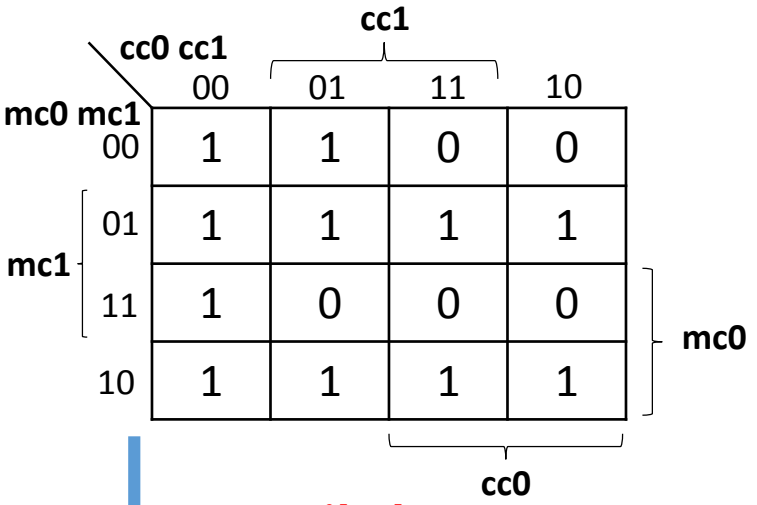
missionary_curr[0] = mc0
missionary_curr[1]= mc1
cannibal_curr[0] = cc0
cannibal_curr[1] = cc1



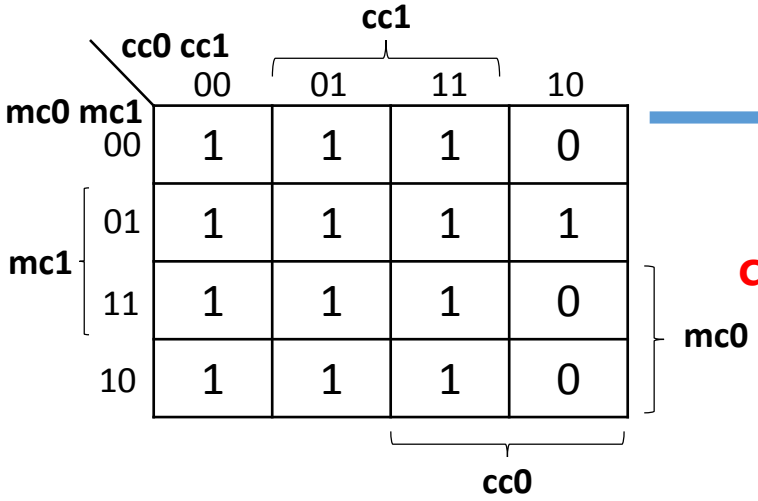
missionary_next[0]



missionary_next[1]



cannibal_next[0]



cannibal_next[1]

missionary_next[0] 의 3차원 K-map

*Direction = D

*D=0

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	0	1	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

mc1 (01, 11), mc0 (01, 11, 10), cc0 (11, 10), cc1 (01, 11)

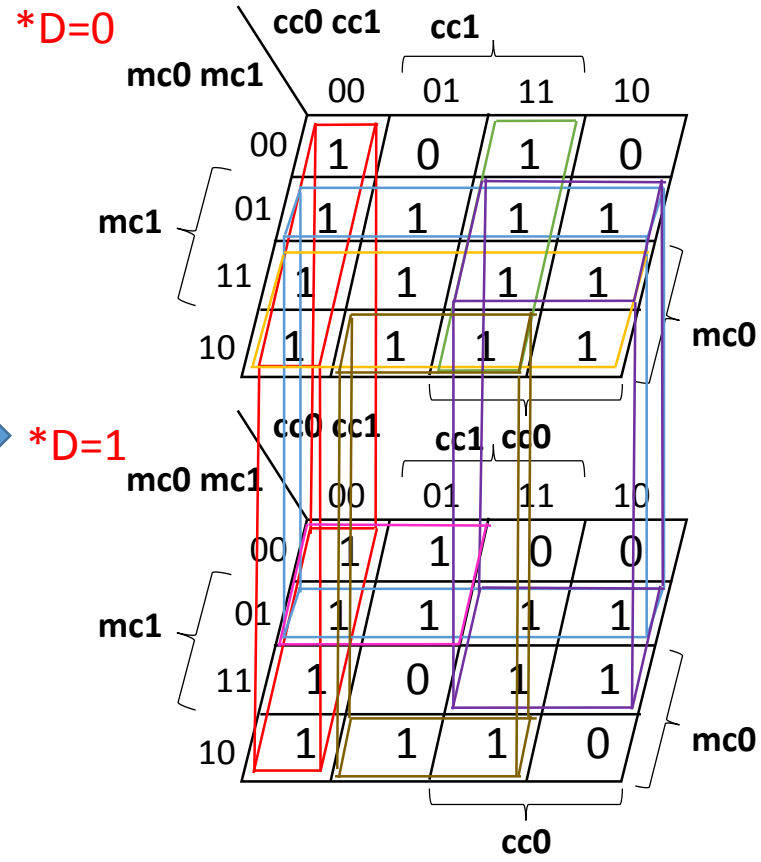
missionary_next[0]

*D=1

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	1	0	1	1
10	1	1	1	0

mc1 (01, 11), mc0 (01, 11, 10), cc0 (11, 10), cc1 (01, 11)

missionary_next[0]



Color	Algebra
Red	cc0'*cc1'
Yellow	D'*mc0
Blue	mc0'*mc1
Purple	cc0*mc1
Green	D'*cc0*cc1
Leaf	cc1*mc0*mc1'
Pink	D*cc0'*mc0'

missionary_next[0] = **cc0'*cc1' + D'*mc0 + mc0*mc1' + cc0*mc1 + D'*cc0*cc1 + cc1*mc0*mc1' + D*cc0'*mc0'**

Global KU Frontier Spirit

missionary_next[1] 의 3차원 K-map

*Direction = D

*D=0

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	0	1	0
01	1	0	1	1
11	1	1	1	1
10	1	1	1	1

mc1 (01, 11), mc0 (11, 10), cc1 (01, 11), cc0 (11, 10)

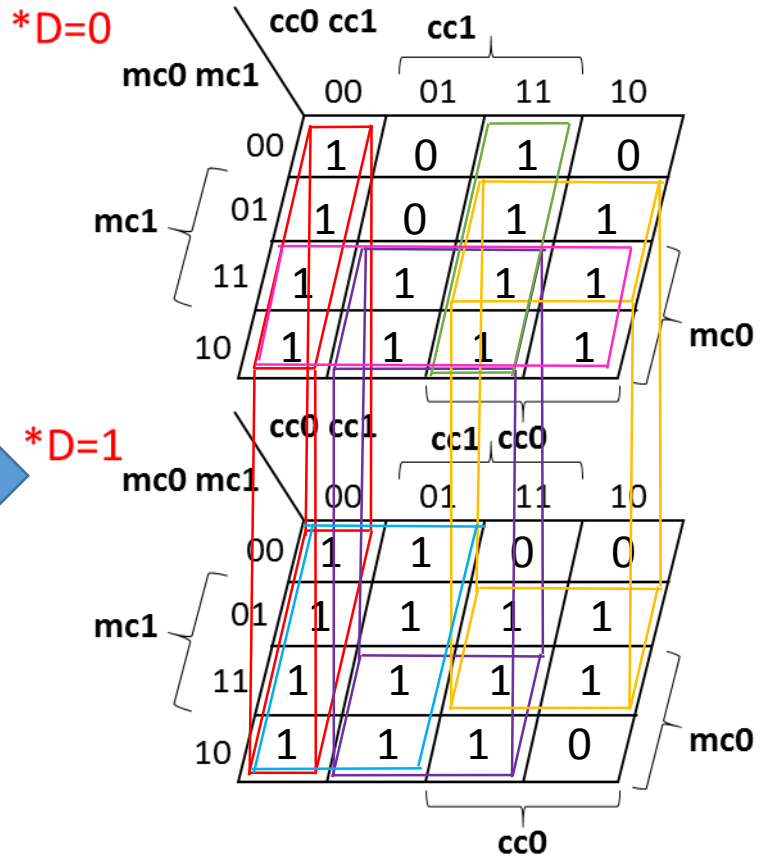
missionary_next[1]

*D=1

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	0	1	0
01	1	0	1	1
11	1	1	1	1
10	1	1	1	1

mc1 (01, 11), mc0 (11, 10), cc1 (01, 11), cc0 (11, 10)

missionary_next[1]



Color	Algebra
Red	cc0'*cc1'
Yellow	cc0*mc1
Blue	D*cc0'
Purple	cc1*mc0
Green	D'*cc0*cc1
Pink	D'*mc0

missionary_next[1] = **cc0'*cc1' + cc0*mc1 + D*cc0' +cc1*mc0 + D'*cc0*cc1 + D'*mc0**

Global KU Frontier Spirit

cannibal_next[0] 의 3차원 K-map

*Direction = D

*D=0

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	1	1	1
01	1	1	1	1
11	0	1	1	1
10	1	1	1	1

mc1 (01, 11), mc0 (00, 01, 11, 10), cc1 (01, 11), cc0 (00, 01)

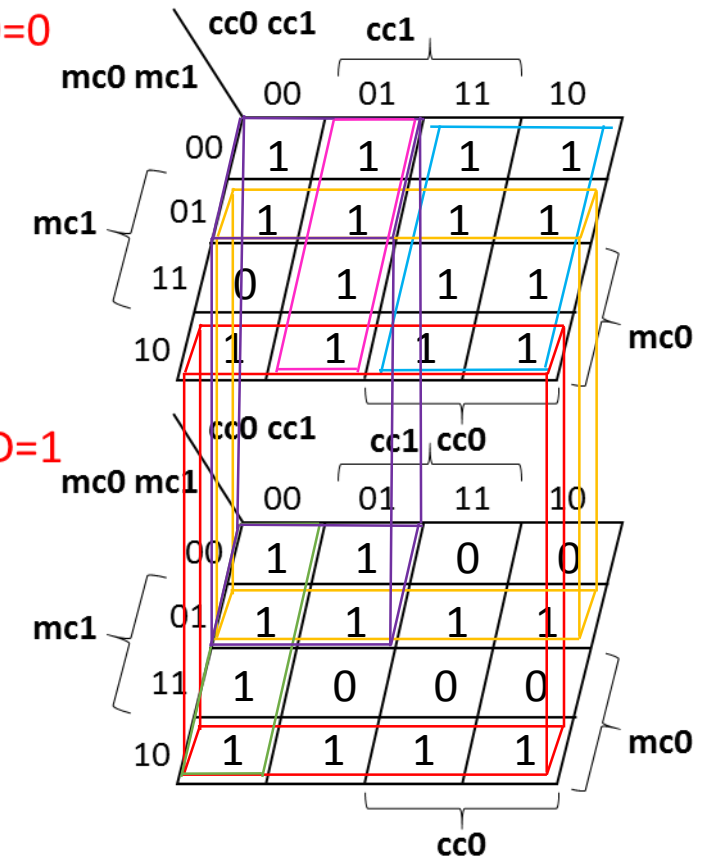
cannibal_next[0]

*D=1

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	1	0	0
01	1	1	1	1
11	1	0	0	0
10	1	1	1	1

mc1 (01, 11), mc0 (00, 01, 11, 10), cc1 (01, 11), cc0 (00, 01)

cannibal_next[0]



Color	Algebra
Red	mc0*mc1’
Yellow	mc0’*mc1
Blue	D’*cc0
Purple	cc0’*mc0’
Green	D*cc0’*cc1’
Pink	D’*cc0’*cc1

cannibal_next[0] = mc0*mc1’ + mc0’*mc1 + D’*cc0 + cc0’*mc0’ + D*cc0’*cc1’ + D’*cc0’*cc1

Global KU Frontier Spirit

cannibal_next[1] 의 3차원 K-map

*D=0

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	0	1	1
01	1	0	1	1
11	1	0	1	1
10	1	1	1	1

mc1 (01, 11), mc0 (11, 10), cc0 (00, 01), cc1 (01, 11, 10)

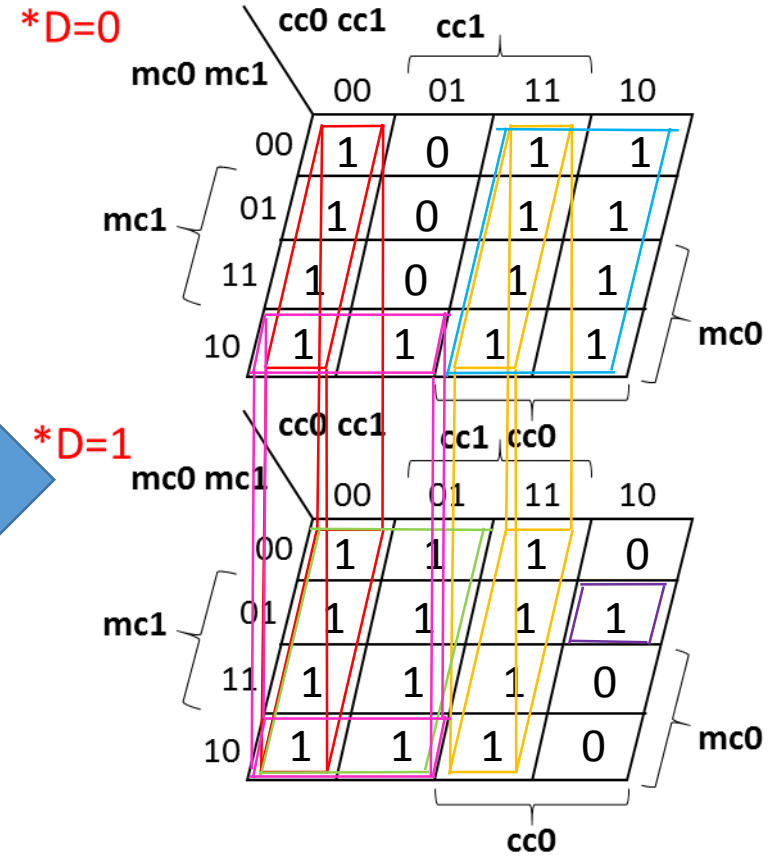
cannibal_next[1]

*D=1

mc0 mc1 \ cc0 cc1	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	1	1	1	0
10	1	1	1	0

mc1 (01, 11), mc0 (11, 10), cc0 (00, 01), cc1 (01, 11, 10)

cannibal_next[1]



*Direction = D

Color	Algebra
Red	cc0'*cc1'
Yellow	cc0*cc1
Blue	D'*cc0
Purple	D*cc0*cc1'*mc0'*mc1
Green	D*cc0'
Pink	cc0'*mc0*mc1'

cannibal_next[1] = **cc0'*cc1' + cc0*cc1 + D'*cc0 + D*cc0*cc1'*mc0'*mc1 + D*cc0' + cc0'*mc0*mc1'**

Global KU Frontier Spirit

K-map을 이용한 Module 구조 파악

Input : cc0 = A[0], cc1 = A[1], mc0 = B[0], mc1 = B[1], D
Output : **missionary_next[0] = H[0], missionary_next[1] = H[1], cannibal_next[0] = I[0], cannibal_next[1] = I[1]**

Gate Level 1 : 오른쪽 그래프(모든 곱의 논리의 통계)를 통해 AND Gate 17개를 통해서 논리의 곱의 계산

Gate Level 2 : OR Gate 4개를 이용하여 각각의 Output을 도출

포트 리스트를 제외하고 필요한 wire 포트신호타입의 수:
AND Gate 17개 = 총 17개

AND Gate 반복 케이스	AND Gate 개수	wire이름	OR Gate 경로
B[0], B[1]	17	aw18	I[1]
B[0], A[1]	16	aw17	H[0]
B[0], A[1]	16	aw17	H[1]
~B[0], ~B[1]	15	aw16	H[0]
~B[0], ~B[1]	15	aw16	H[1]
~B[0], ~B[1]	15	aw16	I[1]
~B[0], A[0], ~A[1]	14	aw15	I[1]
~B[0], ~A[0]	13	aw14	I[0]
B[1], A[0]	12	aw13	H[1]
B[1], A[0], ~A[1]	11	aw12	H[0]
D, B[0], ~B[1], ~A[0], A[1]	10	aw10	I[1]
D, ~B[0]	9	aw9	H[1]
D, ~B[0]	9	aw9	I[1]
D, ~B[0], ~B[1]	8	aw8	I[0]
D, ~B[0], ~A[0]	7	aw7	H[0]
~D, B[0]	6	aw6	I[0]
~D, B[0]	6	aw6	I[1]
~D, B[0], B[1]	5	aw5	H[0]
~D, B[0], B[1]	5	aw5	H[1]
~D, ~B[0], B[1]	4	aw4	I[0]
~D, A[0]	3	aw3	H[0]
~D, A[0]	3	aw3	H[1]
A[0], ~A[1]	2	aw2	I[0]
~A[0], A[1]	1	aw1	I[0]
~A[0], A[1]	1	aw1	H[0]

HDL code

(주석과 함께 2페이지)

HDL Code (Pg 1)

```
module missionary_carnibal_combinational_circuit(H, I, A, B, D);
```
//모듈헤더 파라미터 리스트 생략 포트리스트로 포트신호의 이름을 나열 INPUT 과 OUTPUT 나열

```
    //I/O ports declaration
```
//wire 포트신호타입을 생략하여 포트신호의 방향은 input으로 이름은 모듈 헤더의 포트리스트에서 가져왔다

```
        input [1:0] A;
```
//현재 상태의 선교사의 수를 [1:0]로 Input 포트 신호 방향으로 선언. []는 어레이 선언을 의미한다, [1:0]일 경우 0번 칸에서 1번 칸까지 있는 어레이(총 2칸)를 뜻한다.

```
        input [1:0] B;
```
//현재 상태의 식인종의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        input D;
```
//이동방향을 Input 포트 신호 방향으로 선언

```
        output [1:0] H;
```
//다음 상태의 선교사의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        output [1:0] I;
```
//다음 상태의 식인종의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        //inner net definition
        wire aw1, aw2, aw3, aw4, aw5, aw6, aw7, aw8, aw9, aw10, aw11, aw12, aw13, aw14, aw15, aw16, aw17;
```
//and gate 와 or or gate를 연결하기 위해 and gate의 결과값을 전달하는 wire 포트 타입을 번호 순으로 1부터 17까지 선언

HDL Code (Pg 2)

```
//primitive logic gate instantiation
 //and gate
 //미리 K-map과 테이블로 계산된 값들을 and 게이트를 이용하여 지정 wire 포트 타입의 aw으로 전달한다. (gate level 1)

 and (aw1, ~A[0], A[1]);
 and (aw2, A[0], ~A[1]);
 and (aw3, ~D, A[0]);
 and (aw4, ~D, ~B[0], B[1]); //3개의 input 1개의 결과값
 and (aw5, ~D, B[0], B[1]);
 and (aw6, ~D, B[0]);
 and (aw7, D, ~B[0], ~A[0]);
 and (aw8, D, ~B[0], ~B[1]);
 and (aw9, D, ~B[0]);
 and (aw10, D, B[0], ~B[1], ~A[0], A[1]); // 5개의 input 1개의 결과값
 and (aw11, B[1], A[0], ~A[1]);
 and (aw12, B[1], A[0]);
 and (aw13, ~B[0], ~A[0]);
 and (aw14, ~B[0], A[0], ~A[1]);
 and (aw15, ~B[0], ~B[1]);
 and (aw16, B[0], A[1]);
 and (aw17, B[0], B[1]);

 //or gate
 //미리 K-map과 테이블로 계산된 값들을 or 게이트를 이용하여 output에 전달한다. (gate level 2)
 or (H[0], aw1,aw3,aw5,aw7,aw11,aw15,aw16); //다음 단계의 선교사 수의 첫번째 비트
 or (H[1], aw3,aw5,aw9,aw12,aw15,aw16); //다음 단계의 선교사 수의 두번째 비트
 or (I[0], aw1,aw2,aw4,aw6,aw8,aw13); //다음 단계의 식인종 수의 첫번째 비트
 or (I[1], aw6,aw9,aw,10,aw14,aw15,aw17); //다음 단계의 식인종 수의 두번째 비트

endmodule //모듈의 끝
```

추가 모듈의 구성

A. 추가 모듈 구조 파악

B. HDL Code

추가 모듈 구조 파악

추가 모듈의 역할은 3가지 이다.

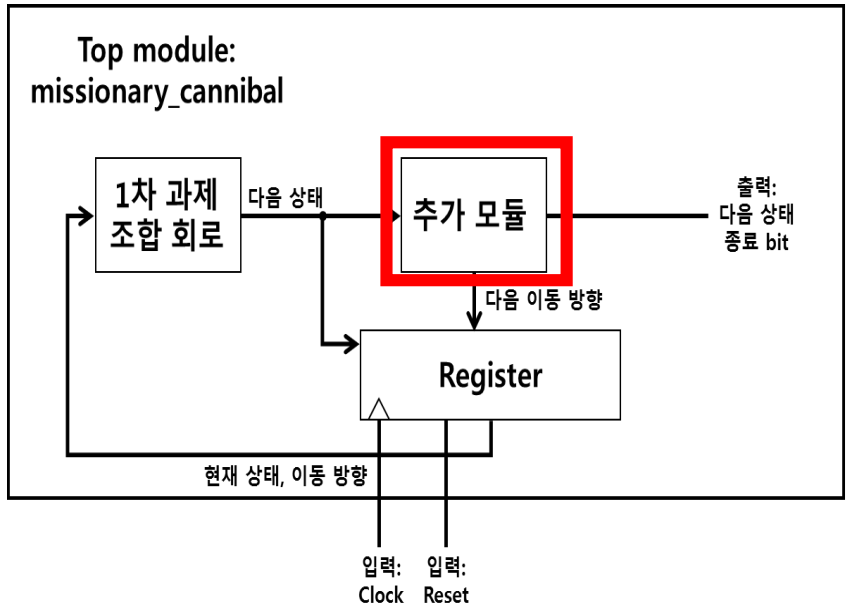
1. 1차 과제 조합회로에서 받은 선교사와 식인종 수의 다음 상태를 받아 그대로 출력해 낸다.
2. 12번 동안 반복되는 해답의 한번의 루트 마다 종료사인인 1bit를 출력해 낸다.
3. 레지스터에 다음 이동방향을 전달한다.

Input : **missionary_next[0] = H[0], missionary_next[1] = H[1], cannibal_next[0] = I[0], cannibal_next[1] = I[1], C**
Output : **L[0], L[1], M[0], M[1], J, K**

여기서 [1:0] H, I; 는 조합회로에서 받은 선도사와 식인종 수의 다음 상태이며 [1:0] L, M;는 이를 아무런 값의 변화 없이 그대로 전달해 주는 변수이다.

C는 Register에서 나오는 이동방향으로 이동방향은 항상 다음 상태로 갈 때 보수가 되므로 J변수에 보수를 취해서 넣어준다.

K는 종료를 알리는 1bit로 다음상태가 0000 즉, K-map 상으로 ~H[1]* ~H[0]* ~I[1]*~I[0]에 1을 반환함으로 **K**를 AND gate하나로 묶어 코딩한다.



HDL Code

```
module Additional_module(L, M, J, K, H, I, C);
  input [1:0] H, I;   //combinational circuit output
  input C;            //register movement direction output
  output [1:0] L, M;
  output J, K;

//H를 L에 I를 M에 직접연결
  assign L[1] = H[0];
  assign L[0] = H[1];
  assign M[1] = I[0];
  assign M[0] = I[1];

  assign J = (~C);    //next movement direction
  and(K, ~H[1], ~H[0], ~I[1], ~I[0]);    //finish bit

 endmodule
```

Register 모듈의 구성

A. Register 모듈 구조 파악

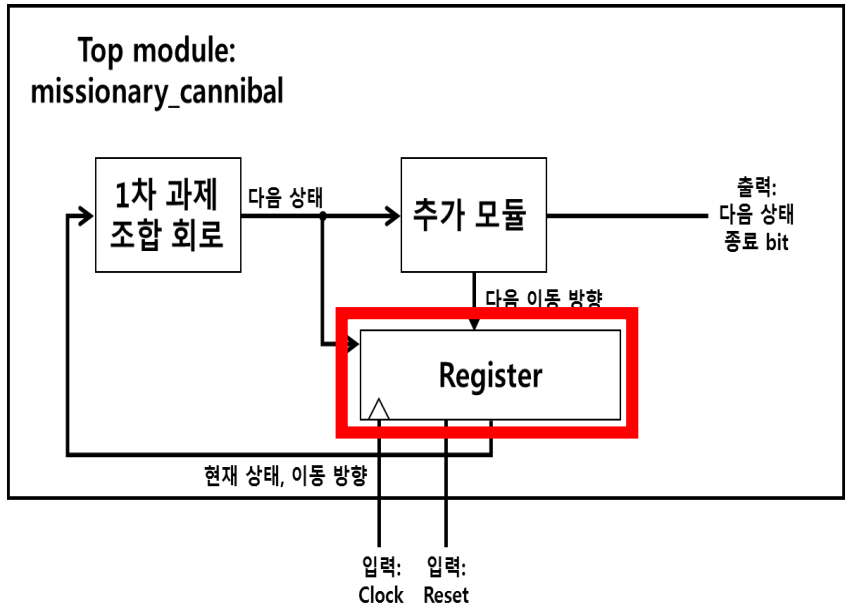
B. HDL Code

Register 모듈 구조 파악

Register 모듈의 역할은 1가지 이다. 하지만 이 한가지 역할은 전체 모듈에서 제일 중요한 핵심이다.

Register 모듈은 추가모듈로부터 다음 이동방향을 받고 1차 조합회로에서의 다음 상태를 받아 여러 개의 D 플립플랍에 저장하고 이를 Clock과 Reset의 인풋을 받아 특정 시간의 변화에 순차적으로 변화한 현재 상태를 다시 1차 과제 조합회로로 전달한다.

이 과정에서 단순히 인풋에 따라 달라지는 1차 과제의 조합회로와 다르게 클락에 따라 변화하는 조합회로의 아웃풋을 인풋으로 받아 다시 그 조합회로의 인풋으로 전달하여 순차적인 변화를 만들어 낸다.



HDL Code

```
//D flip-flop with asynchronous reset.
module D_Flip_Flop(Q,D,CLK,RST);
  output Q;
  input D, CLK, RST;
  reg Q;
  always @(posedge CLK or negedge RST)
    if (~RST) Q = 1'b0; // Same as : if (RST==0)
    else Q = D;
endmodule
```

헤더 모듈의 구성

A. 헤더 모듈 구조 파악

B. HDL Code

헤더 모듈 구조 파악

앞서 정의했던 D 플립플랍, 1차 과제 조합회로, 추가 모듈을 불러와 이들을 연결해 2차 과제의 최종 목적인 회로를 구현한다.

클락과 리셋을 인풋으로 받으며 매 클럭 사이클마다 5개의 레지스터들에 저장되어 있던 선교사와 식인종의 현재 숫자, 이동 방향을 wire A, B, D를 통해 1차 과제 조합회로로 전달한다.

이때 이동 방향은 추가모듈에도 동시에 전달된다.

조합회로에서 계산된 다음 상태는 와이어 H, I를 통해 추가 모듈과 레지스터로 전달된다.

레지스터의 인풋으로 들어간 다음 상태는 저장되었다가 다시 1차과제 조합회로로 전달된다.

추가모듈은 다음상태와 레지스터에서 전달된 이동방향을 받아 다음 이동방향의 값, 종료여부를 계산하고 종료 시 1을, 그 외의 경우 0을 종료비트로 출력하는 동시에 다음 이동방향을 레지스터로 전달한다.

추가모듈에서 출력되는 다음상태(L, M)와 종료비트(K)가 헤더 모듈의 아웃풋이 된다.

HDL Code

```
module missionary_carnibal_sequential_circuit(L, M, K, clock, reset);
 output [1:0] L, M;
 output K;
 input clock, reset;
 wire [1:0] A, B, H, I;
 wire D, J;

 D_Flip_Flop  FF0 (A[0], H[0], clock, reset),
              FF1 (A[1], H[1], clock, reset),
              FF2 (B[0], I[0], clock, reset),
              FF3 (B[1], I[1], clock, reset),
              FF4 (D, J, clock, reset);        //D flip flop 5개 (FF0~FF4) load

 missionary_carnibal_combinational_circuit     CC0 (H, I, A, B, D);   //1차 과제 조합회로 load

 Additional_module                             AM0 (L, M, J, K, H, I, D);  //추가 모듈 AM0 load

 endmodule
```

Clock Cycle Time을 줄이는 방법

A. Clock Time 측정 방법
B. 두가지 다른 조합회로의 Clock Time 비교
① 디코더를 이용한 전체 모듈의 최소 Clock Time
② 최소화된 Gate Level를 이용한 전체 모듈의 최소 Clock Time
C. 결과적인 Clock Cycle Time

Clock Time 측정 방법

1. Clock Time의 중요성
   - VHDL 합성에 있어서 타이밍과 동작기능은 항상 같이 고려해야 할 문제이다. 설계자가 설계한 것을 시뮬레이션을 통하여 정상적인 동작을 할 것인지 검증한 다음, 합성과정을 거치게 된다. 이러한 합성과정에서, VHDL 코드는 특정한 Technology library(최종 제작할 칩의 공정에 해당하는 셀라이브러리)에 있는 하드웨어 로직게이트로 매핑된다. 만일 합성결과가 타이밍 기준을 만족시키지 않으면 설계자는 설계내용을 분석하고 타이밍 문제를 다시 해결하여야 합니다. 일반적으로 타이밍 위반(timiming violation)은 두가지가 있으며, 하나는 Setup timing violation이며 또하나는 Hold timing violation입니다.
     - Setup timing violation : 이는 합성된 결과가 클럭 주기보다도 큰 딜레이를 갖는 combinational logic(조합회로)를 갖아 생기는 문제
     - Hold timing violation : Hold time violation은 너무빠르게 설계되었을 때 주로 발생, 예를 들어 상승에지에 동작하는 플립플롭이 입력값을 인지하기도 전에 입력이 바뀌면 hold time violation이 발생
2. 합성에 있어서 Setup/Hold timing 의 고려
   - 합성을 완료한 후 설계자는 합성된 결과를 가지고 정적 타이밍 분석을 하여, setup 또는 hold time violation이 있는 지 확인.
   - Violation이 없는 최소의 타이밍을 찾는 것이 목표
   - **BUT! 쿼터스에서는 Slack, Setup time과 Hold time을 계산하기 어렵다. 그래서 우린 clock의 측정 방법을 “K”종료비트가 제대로 출력이 되는지 안되는지의 여부로 파악하였다.**

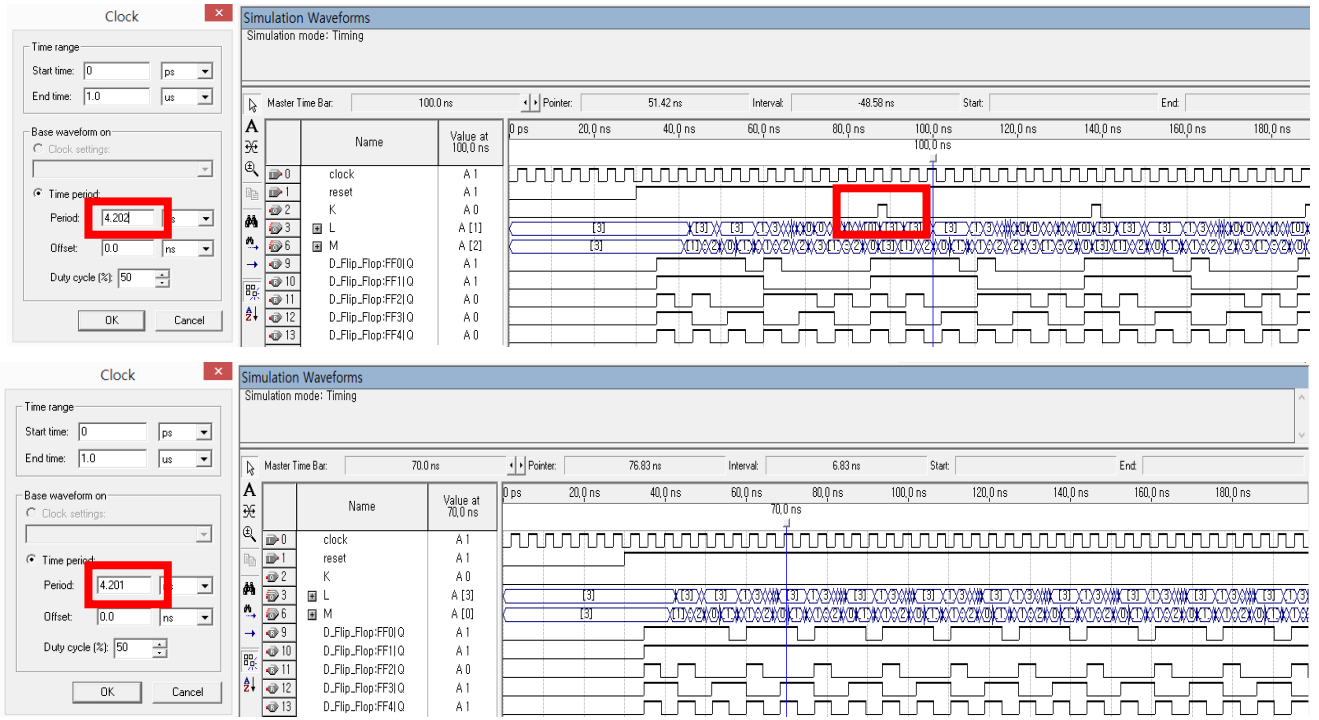
디코더를 이용한 전체 모듈의 최소 Clock Time

조합회로에 현재 적용되어 있는 Gate Level과 똑같은 INPUT과 똑같은 OUTPUT을 같는 디코더를 넣으면 전체 모듈의 시뮬레이션 결과 값은 똑같이 나온다.

하지만, CLOCK CYCLE TIME에서는 확연한 차이를 보인다.

이의 예시를 오른쪽에서 볼 수 있다.

```
1  module decoder_2x4_gates(D, A);  //instantiate 2x4 decoder
2    output   [0:3] D;
3    input    [1:0] A;
4    wire     A1, A0;
5
6    not
7     G1 (A1, A[1]),
8     G2 (A0, A[0]);
9
10   and
11    G3 (D[0], A1, A0),
12    G4 (D[1], A1, A[0]),
13    G5 (D[2], A[1], A0),
14    G9 (D[3], A[1], A[0]);
15 endmodule
16
17 module decoder_3x8_gates(D, A, B, E);  //instantiate 3x8 decoder
18   output   [0:7] D;
19   input    [1:0] A;
20   input    B;
21   input    E;
22   wire     A1, A0, B0;
23
24   not
25   G1 (A1, A[1]),
26   G2 (A0, A[0]),
27   G3 (B0, B);
28
29   and
30   G4 (D[0], A1, A0, B0, E),
31   G5 (D[1], A1, A0, B, E),
32   G6 (D[2], A1, A[0], B0, E),
33   G7 (D[3], A1, A[0], B, E),
34   G8 (D[4], A[1], A0, B0, E),
35   G9 (D[5], A[1], A0, B, E),
36   G10 (D[6], A[1], A[0], B0, E),
37   G11 (D[7], A[1], A[0], B, E);
38 endmodule
```



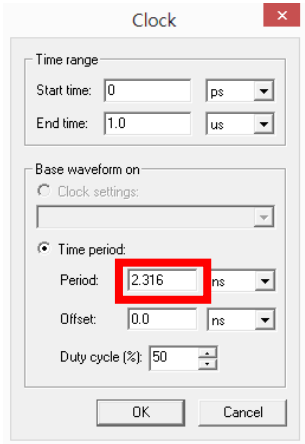
0.001 ns차이로 K의 아웃풋이 나오지 않는다. 즉 디코더를 사용한 모듈의 최소 Clock Cycle Time은 4.202ns라고 할 수 있다.

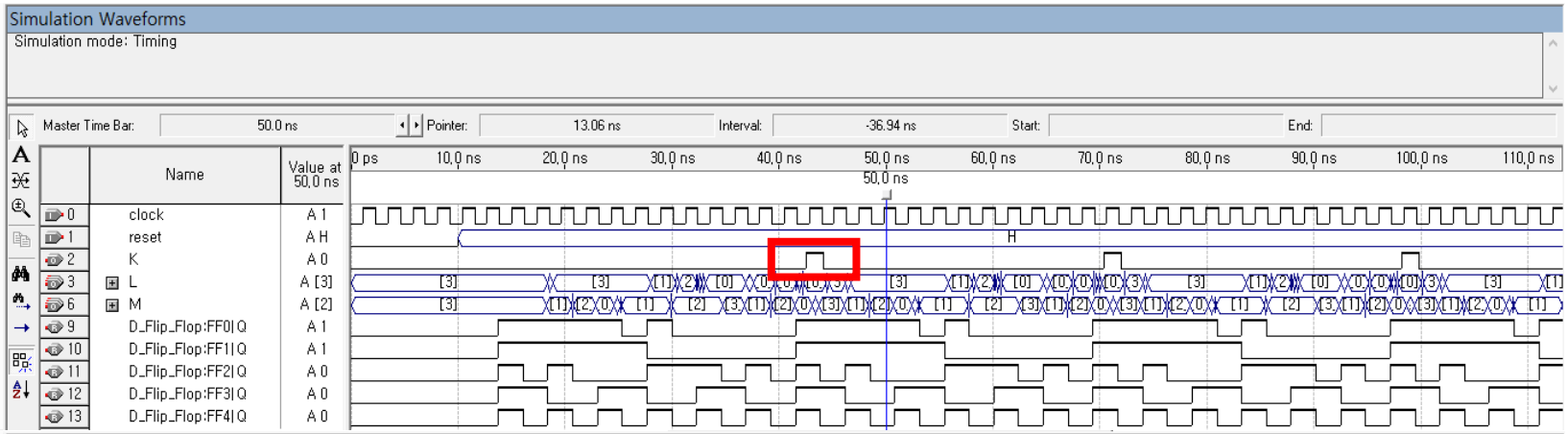
이는 최소화된 게이트 레벨을 사용한 모듈과 확연한 차이가 난다. (다음페이지에서 비교)

최소화된 Gate Level를 이용한 전체모듈의 최소 Clock Time

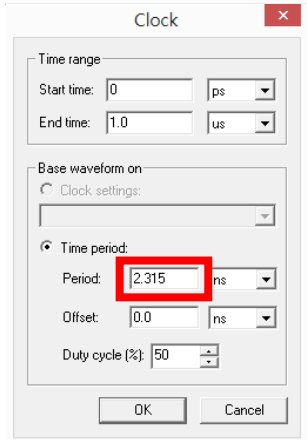
최소화된 Gate Level를 사용한 현재 적용중인 전체 모듈의 Clock Cycle Time을 알아보자.

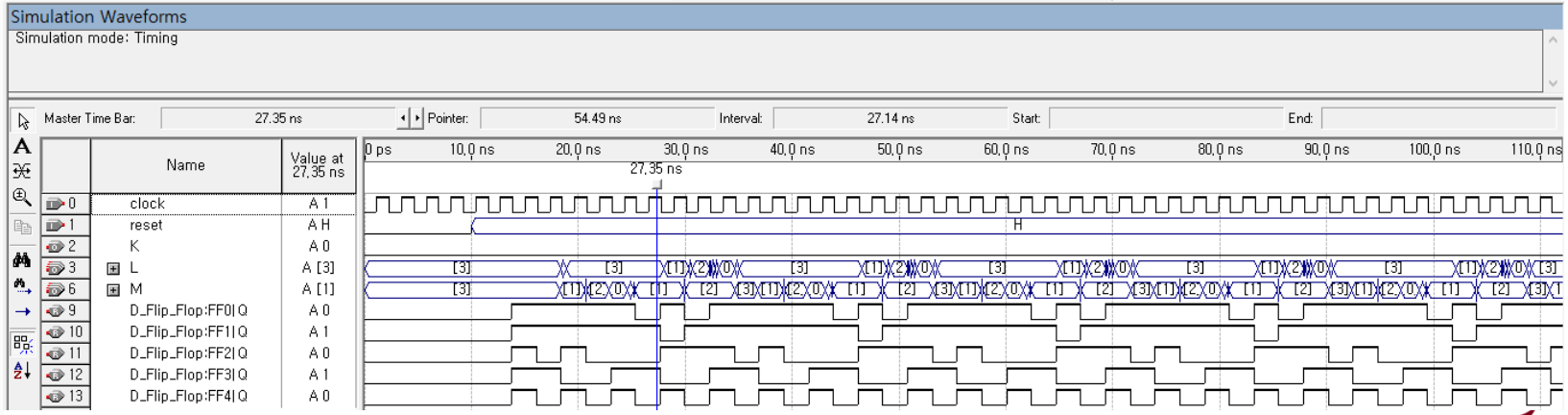
0.001 ns차이로 K의 아웃풋이 나오지 않는다. 즉 최소화된 Gate Level 을 사용한 모듈의 최소 Clock Cycle Time은 2.316ns라고 할 수 있다.





이를 이전에 디코더를 사용한 전체 모듈의 최소 Clock Cycle Time과 비교하자면





디코더	최소화 Gate level
4.202ns	> 2.316ns

결론:

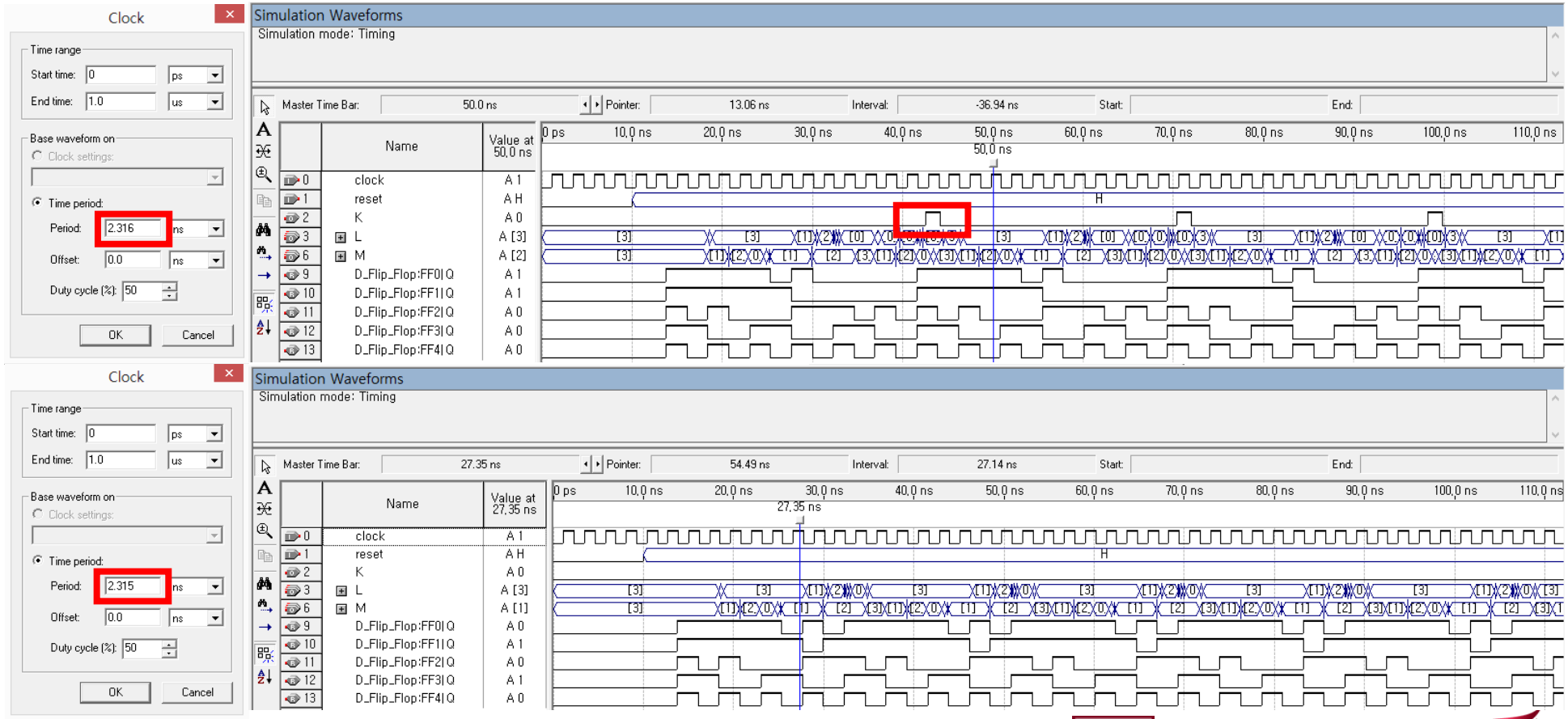
디코더가 확연히 느리다.

이는 디코더가 더 큰 딜레이를 갖는 combinational logic = **Setup time** 이 늘어나기 때문

Global KU Frontier Spirit

결과적인 Clock Cycle Time

0.001 ns차이로 K의 아웃풋이 나오지 않는다. 즉 최소화된 Gate Level 을 사용한 모듈의 최소 Clock Cycle Time은 2.316ns라고 할 수 있다.



최종 Clock Cycle Time

2.316ns

HDL code

(주석과 함께 5페이지)

HDL Code (Pg 1)

```
module missionary_carnibal_combinational_circuit(H, I, A, B, D);
```
//모듈헤더 파라미터 리스트 생략 포트리스트로 포트신호의 이름을 나열 INPUT 과 OUTPUT 나열

```
    //I/O ports declaration
```
//wire 포트신호타입을 생략하여 포트신호의 방향은 input으로 이름은 모듈 헤더의 포트리스트에서 가져왔다

```
        input [1:0] A;
```
//현재 상태의 선교사의 수를 [1:0]로 Input 포트 신호 방향으로 선언. []는 어레이 선언을 의미한다, [1:0]일 경우 0번 칸에서 1번 칸까지 있는 어레이(총 2칸)를 뜻한다.

```
        input [1:0] B;
```
//현재 상태의 식인종의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        input D;
```
//이동방향을 Input 포트 신호 방향으로 선언

```
        output [1:0] H;
```
//다음 상태의 선교사의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        output [1:0] I;
```
//다음 상태의 식인종의 수를 [1:0]로 Input 포트 신호 방향으로 선언

```
        //inner net definition
        wire aw1, aw2, aw3, aw4, aw5, aw6, aw7, aw8, aw9, aw10, aw11, aw12, aw13, aw14, aw15, aw16, aw17;
```
//and gate 와 or or gate를 연결하기 위해 and gate의 결과값을 전달하는 wire 포트 타입을 번호 순으로 1부터 17까지 선언

HDL Code (Pg 2)

```
//primitive logic gate instantiation
 //and gate
 //미리 K-map과 테이블로 계산된 값들을 and 게이트를 이용하여 지정 wire 포트 타입의 aw으로 전달한다. (gate level 1)

 and (aw1, ~A[0], A[1]);
 and (aw2, A[0], ~A[1]);
 and (aw3, ~D, A[0]);
 and (aw4, ~D, ~B[0], B[1]); //3개의 input 1개의 결과값
 and (aw5, ~D, B[0], B[1]);
 and (aw6, ~D, B[0]);
 and (aw7, D, ~B[0], ~A[0]);
 and (aw8, D, ~B[0], ~B[1]);
 and (aw9, D, ~B[0]);
 and (aw10, D, B[0], ~B[1], ~A[0], A[1]); // 5개의 input 1개의 결과값
 and (aw11, B[1], A[0], ~A[1]);
 and (aw12, B[1], A[0]);
 and (aw13, ~B[0], ~A[0]);
 and (aw14, ~B[0], A[0], ~A[1]);
 and (aw15, ~B[0], ~B[1]);
 and (aw16, B[0], A[1]);
 and (aw17, B[0], B[1]);

 //or gate
 //미리 K-map과 테이블로 계산된 값들을 or 게이트를 이용하여 output에 전달한다. (gate level 2)
 or (H[0], aw1,aw3,aw5,aw7,aw11,aw15,aw16); //다음 단계의 선교사 수의 첫번째 비트
 or (H[1], aw3,aw5,aw9,aw12,aw15,aw16); //다음 단계의 선교사 수의 두번째 비트
 or (I[0], aw1,aw2,aw4,aw6,aw8,aw13); //다음 단계의 식인종 수의 첫번째 비트
 or (I[1], aw6,aw9,aw,10,aw14,aw15,aw17); //다음 단계의 식인종 수의 두번째 비트

endmodule //모듈의 끝
```

HDL Code (Pg 3)

```
module Additional_module(L, M, J, K, H, I, C);
  input [1:0] H, I;   //combinational circuit output
  input C;            //register movement direction output
  output [1:0] L, M;
  output J, K;

//H를 L에 I를 M에 직접연결
  assign L[1] = H[0];
  assign L[0] = H[1];
  assign M[1] = I[0];
  assign M[0] = I[1];

  assign J = (~C);    //next movement direction
  and(K, ~H[1], ~H[0], ~I[1], ~I[0]);    //finish bit

 endmodule
```

HDL Code (Pg 4)

```
//D flip-flop with asynchronous reset.
module D_Flip_Flop(Q,D,CLK,RST);
  output Q;
  input D, CLK, RST;
  reg Q;
  always @(posedge CLK or negedge RST)
    if (~RST) Q = 1'b0; // Same as : if (RST==0)
    else Q = D;
endmodule
```

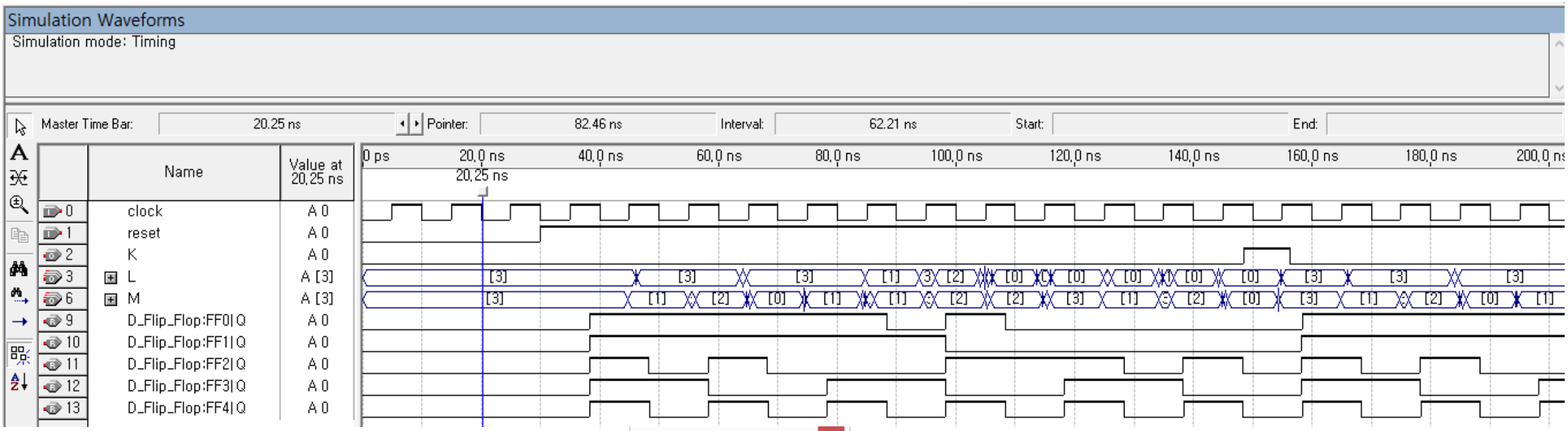
HDL Code (Pg 5)

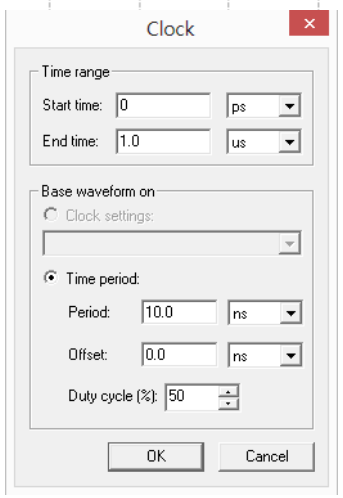
```
module missionary_carnibal_sequential_circuit(L, M, K, clock, reset);
 output [1:0] L, M;
 output K;
 input clock, reset;
 wire [1:0] A, B, H, I;
 wire D, J;

 D_Flip_Flop  FF0 (A[0], H[0], clock, reset),
              FF1 (A[1], H[1], clock, reset),
              FF2 (B[0], I[0], clock, reset),
              FF3 (B[1], I[1], clock, reset),
              FF4 (D, J, clock, reset);        //D flip flop 5개 (FF0~FF4) load

 missionary_carnibal_combinational_circuit     CC0 (H, I, A, B, D);   //1차 과제 조합회로 load

 Additional_module                             AM0 (L, M, J, K, H, I, D);  //추가 모듈 AM0 load

 endmodule
```

Simulation screenshot

1. Clock Cycle Time 이 10ns인 시뮬레이션
2. Clock Cycle Time 이 최소인 시뮬레이션

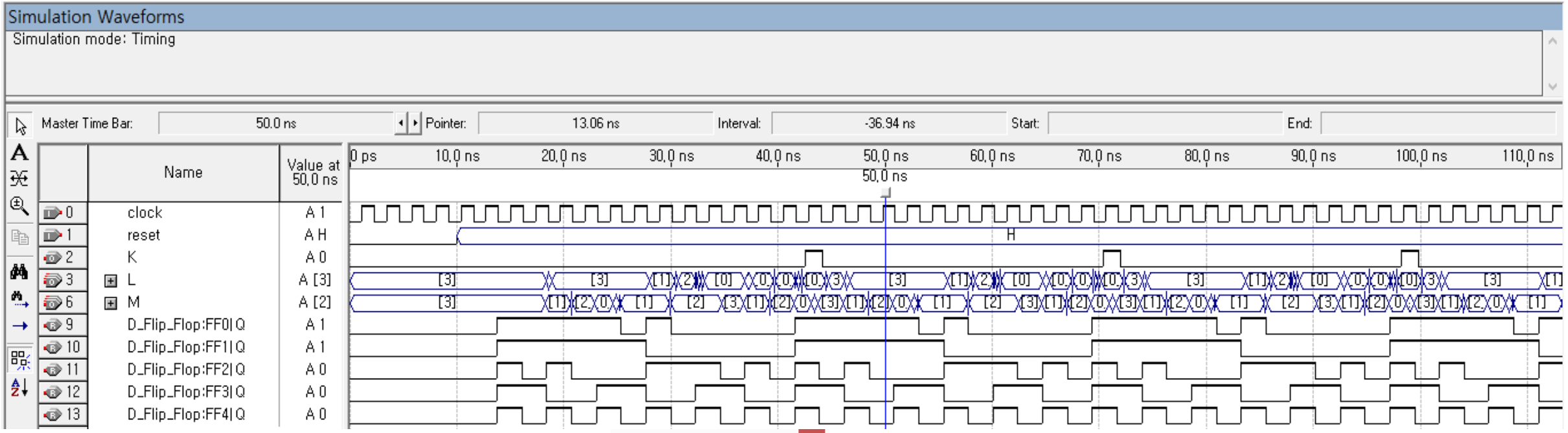
1. Clock Cycle Time 이 10ns인 시뮬레이션

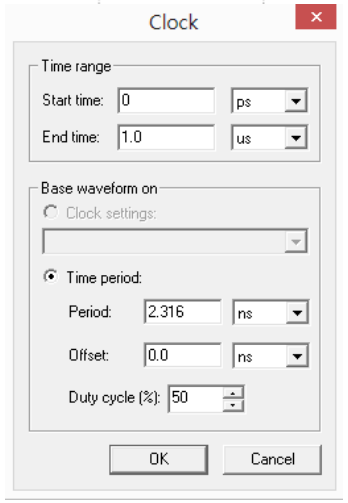




Global KU Frontier Spirit

2. Clock Cycle Time 이 최소인 시뮬레이션





Global KU Frontier Spirit

논리설계 텀 no.2 끝

Moon Gi Jun 2013210063

정주혜 2015320142